JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Mercury: a Model for Live Remote Debugging in Reflective Languages

Nick Papoulias[a] Noury Bouraqadi[b] Luc Fabresse[b]
Stéphane Ducasse[a] Marcus Denker[a]

a. RMoD, Inria Lille Nord Europe, France
http://rmod.lille.inria.fr

b. Mines Telecom Institute, Mines Douai
http://car.mines-douai.fr/

Abstract Remote debugging facilities are a technical necessity for devices that have limited computing power to run an IDE (*e.g.,* smartphones), lack appropriate input/output interfaces (display, keyboard, mouse) for programming (e.g mobile robots) or are simply unreachable for local development (e.g cloud-servers). Yet remote debugging solutions can prove awkward to use due to their distributed nature. Empirical studies show us that on average 10.5 minutes per coding hour (over five 40-hour work weeks per year) are spend for re-deploying applications while fixing bugs or improving functionality. Moreover current solutions lack facilities that would otherwise be available in a local setting because its difficult to reproduce them remotely. Our work identifies three desirable properties that an ideal solution for remote debugging should exhibit, namely: *run-time evolution, semantic instrumentation and adaptable distribution*. Given these properties we propose and validate Mercury, a live remote debugging model and architecture for reflective OO languages.

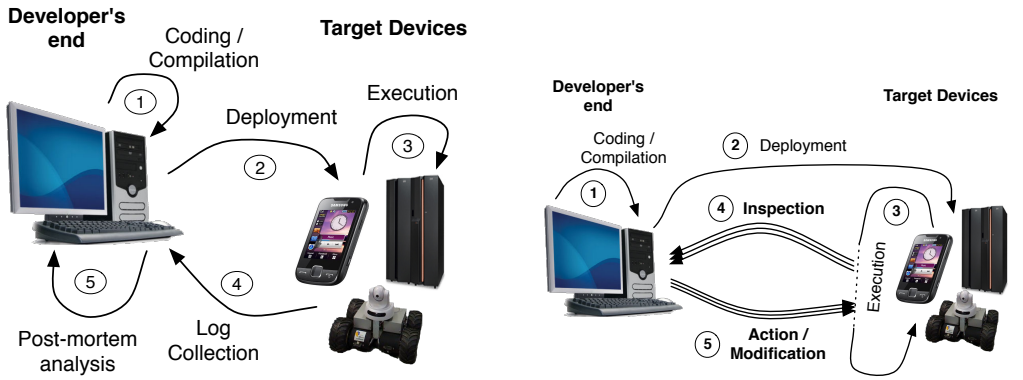
Keywords Remote Debugging, Reflection, Mirrors, Run-Time Evolution, Semantic Instrumentation, Adaptable Distribution, Agile Development

1 Introduction

More and more of our computing devices cannot support an IDE (such as our smartphones or tablets) either due to resource constraints or because they lack input/output interfaces (keyboard, mouse or screen) for development (*e.g.,* robots). Remote debugging is a technical necessity in these situations since targeted devices have different hardware or environment settings than development machines.

Yet remote debuggers can prove awkward to use due to their distributed nature. One such example is the cost of re-deployments in-between remote debugging sessions. Empirical studies show us that on average 10.5 minutes per coding hour (over five 40-hour work weeks

Nick Papoulias, Noury Bouraqadi, Luc Fabresse, Stéphane Ducasse, Marcus Denker. *Mercury: a Model for Live Remote Debugging in Reflective Languages.* In Journal of Object Technology, vol. V, no. N, 2011, pages M:1–22.
doi:10.5381/jot.201Y.VV.N.aN



(a) Debugging with *Post-Mortem* Analysis

(b) Remote debugging and interaction with a live execution

Figure 1 – Log-based Static Debugging vs. Remote Debugging

per year) are spent for re-deploying applications while fixing bugs or improving functionality [Zer11]. This means that the specific facilities that a remote debugging solution offers (e.g. for incremental updating or experimentation) during a remote debugging session can have a huge impact on productivity.

Moreover current solutions lack facilities that would otherwise be available in a local setting because its difficult to reproduce them remotely (e.g., object-centric debugging [RBN12]). This fact can impact the amount of experimentation during a remote debugging session - compared to a local setting. Although emulators for target devices can help in this case, they are not always available, and are often of limited accuracy when sensory input or actuators are involved.

Figure 1(a) shows how developers can debug a target in absence of any support. Coding and compilation (step 1 in Figure 1(a)) need to be done on a developer machine, that is supposed to provide an IDE. Once the software is compiled, it is deployed (step 2) and executed (step 3) on the target. Next, the execution log is collected and transferred to the developer's machine (step 4). Last, the developer can perform a *post-mortem* analysis of the execution log (step 5) to find out hints about defect causes.

When a problem arises during execution, the developer only relies on the log verbosity to identify the causes during the *post-mortem* analysis (step 5). If the log is too verbose, the developer might be overwhelmed with the amount of data. Conversely, limited logging requires to go again through a whole cycle, after adapting the code just for collecting more data. This is due to the static nature of logs whose content is determined at the coding and compilation step (step 1). These cycles for re-compilation and re-deployment are time consuming and make debugging awkward.

In Figure 1(b) we show the different steps of a *remote debugging* process. Steps 1 (coding) and 2 (deployment) are the same as before. However, the execution (step 3) is now *interruptible*. This *interruption* is either user-generated (the developer chooses to freeze the execution to inspect it) or is based on predetermined execution events such as exceptions. Steps 4 and 5 represent the remote debugging loop. This loop takes place at execution time and in the presence of the execution context of the problem which can be inspected and modified. Step 4 represents the remote inspection phase, where information about the current execution context is retrieved from the target. While in step 5 we depict the modification phase where the developer can provide further user-generated interruption points (breakpoints, watchpoints, etc.), alter execution and its state (step, proceed, change the values of variables),

or incrementally update parts of the code deployed in step 2 (save-and-continue, hot-code-swapping). Several loops can occur during the execution depending on the developers' actions (step, proceed, user-generated interruptions) and on execution events (exceptions, errors, etc.). Having the ability to introspect and modify a live execution (without loosing the context) is a major advantage compared to analyzing static logs.

In this work we identify three desirable properties for remote debugging: *run-time evolution, semantic instrumentation and adaptable distribution*. We propose a live model for remote debugging that relies on reflection and more specifically on the concept of Mirrors [BU04]. We show how this model can meet the properties we have identified and we present its implementation in the Pharo language [BDN+09]. Finally we validate our proposal by exemplifying remote debugging techniques supported by Mercury's properties, such as *remote agile debugging* and *remote object instrumentation*.

The contributions of this paper are the following:

- The identification of three desirable properties for remote debugging solutions.
- The definition of a remote meta-level and infrastructure for remote debugging that can exhibit these properties.
- A prototype implementation of our model and its validation.

Our work is organized as follows: we first introduce the properties for remote debugging solutions (Section 2) that we have identified. Then in Section 3 - given these properties - we evaluate existing solutions. Section 4 presents our proposed model: Mercury. Section 5 details our prototype implementation of the Mercury model. Section 6 presents the experimental setting and validation of our proposal. Finally Section 7 concludes our work and presents future perspectives.

2 Desirable Properties of Remote Debugging Solutions

In this Section, we present three desirable properties for remote debugging solutions that we have identified, namely: *run-time evolution*, *semantic instrumentation* and *adaptable distribution*. We introduce and discuss each property individually based on a typical software stack for remote debugging.

As we depict in Figure 2 the target device (on the right) that runs the debugged application must provide a middleware for communication and run-time debugging support for examining processes, the execution stack, the system's organization, introspection of instance and local variables, etc.. On the other hand, the developer machine must provide a middleware layer, debugging tools, but also a model of the running application that describes the application running on the target (*e.g.,* source code or breakpoints).

2.1 Run-Time Evolution

Run-time evolution is the ability to *dynamically* inspect and change the target's application code and state. By dynamically here we mean that inspections and changes on the target can be performed while the application is running.

In Figure 2, there is an implicit relationship between the model of the debugged application (on the developer's end), and the state of the debugged application (on the target). This relationship can be either *static* or *dynamic*, depending on whether a change in either one of

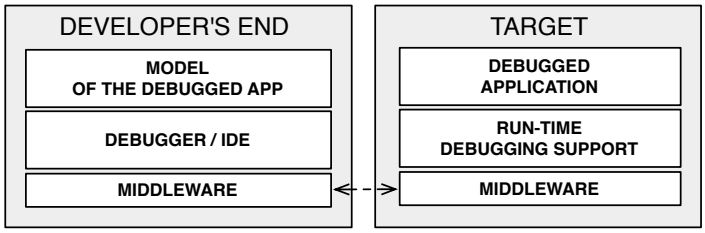


Figure 2 – Software Entities Involved in Remote Debugging.

them updates the other. When a remote debugging solution supports run-time evolution, this relationship is dynamic.

The fact that the target application does not need to be restarted to be debugged and evolved allows developers to:

- Track the origins of bugs and fix them without losing the execution context.
- Fix flaws [Zel05] from within the debugger. Flaws are architectural bugs that are not associated with a specific location in the source file and require an architectural update (removing or adding new code) in order to be addressed.
- Increase productivity while debugging applications with a long startup time.
- Debug critical applications (*e.g.,* server side applications) that cannot be restarted.
- Experiment with heisenbugs [Gra86] as they are observed.

Ideally, in OO languages developers should be able to evolve every organizational module of the target application while debugging. These changes should include:

Add/Rem Packages The ability to introduce new packages (i.e named groups of classes) and remove existing ones.

Add/Rem Classes The ability to introduce new classes and to remove existing ones.

Add/Rem Superclasses The ability to edit a class hierarchy.

Add/Rem Methods The ability to introduce new methods to a class and to edit or remove existing ones.

Add/Rem Fields The ability to introduce new fields to a class or remove existing ones.

2.2 Semantic Instrumentation

With the term *semantic instrumentation* we refer to the ability of a debugging solution to alter the semantics of a running process to assist debugging. Instrumentation is the underlying mechanism through which breakpoints and watchpoints are implemented. A debugging solution *instruments* the running process to halt at specific locations in the code, or when specific events occur (such as variable access) to either return control to the debugging environment or to perform predetermined checks and actions (such as breakpoint conditions).

Ideally in OO languages developers should be able to halt and inspect the running program both at specific locations in the source code and on specific semantical events that involve

objects. In literature these events are referred to as *dynamic reification categories* [RC00]. These categories are a set of operations that can be thought of as *events* which are required for object execution [McA95][RRGN10].

Taking into account these semantic events, instrumentation categories for debugging should at least include:

Statement Execution The ability to halt at a specific statement or line in the source code.

Method Execution The ability to halt at a specific method in the source code upon entry.

Class Instantiation The ability to halt at object creation of specific classes.

Class Field Read The ability to halt when a specific field of *any* instance of a class is read.

Class Field Write The ability to halt when a specific field of *any* instance of a class is written.

Object Read The ability to halt at *any* read attempt on a specific object.

Object Write The ability to halt at *any* write attempt on a specific object.

Object Send The ability to halt at any message send *from* a specific object.

Object Receive The ability to halt at any message send *to* a specific object.

Object as Argument The ability to halt whenever a specific object is passed as an argument.

Object Stored The ability to halt whenever a new reference to a specific object is stored.

Object Interaction The ability to halt whenever two specific objects interact in any way. This is a composite category as defined in [RRGN10]. It can be seen as a composition of object receive, send and argument categories.

2.3 Adaptable Distribution

As seen in Figure 2 remote debugging requires a communication middleware. Ideally a debugging solution should depend on middleware that is extendable and adaptable even at runtime, to address the different communication needs of different targets. For example targets with different resources (memory, processing power, bandwidth) may require different serialization policies. While others such as server applications may require different security policies when they are being debugged through an open network.

We distinguish the following four categories of distribution support for debugging solutions, in ascending order of adaptability:

No-Distribution (-) The debugging solution does not support remote debugging.

Fixed-Middleware (+) The debugging solution supports remote debugging via a dedicated and fixed protocol which cannot be easily extended.

Extensible Middleware (++) The debugging solution supports remote debugging via a general solution for distributed computing (such as an object request broker) which can be extended, such as CORBA or DCOM.

Adaptable Middleware (+++) The debugging solution supports remote debugging via a general solution for distributed computing which can be extended and adapted at runtime [DL02].

3 Evaluation of Existing Solutions

We now study existing debugging solutions of major OO languages in current use today (Java (JPDA) [Ora13b, Ora13a], C# (.NET Debugger) [Mic12a], C++ and Objective-C (through Gdb) [RS03]) as well as dynamic languages with *live programming* support (such as Smalltalk and its debugging model [LP90]), taking also into account bleeding-edge technological achievements [Zer12] and very recent research results [WWS10, RBN12].

3.1 Existing Solutions

JPDA Java's debugging framework stack is JPDA [Ora13b] and it consists of a mirror interface (JDI) [Ora13a, BU04], a communications protocol (JDWP) and the debugging support on the target as part of the virtual-machine's infrastructure (JVM TI). The application on the target machine must be specifically run with debugging support from the VM (the JVM TI) for any interaction between the client and the target to take place. JPDA does not provide facilities to dynamically update the target other then the hot-swapping of pre-existing methods. The communication stress is handled by the low-level debugging communication protocol (JDWP), whose specification is statically defined.

JRebel and DCE The DCE VM [WWS10] and Jrebel [Zer12] are both modifications for the Java virtual machine that support redefinition of loaded classes at runtime. Although these modifications of the underlying VM are not a solution for debugging themselves, they do provide incremental updating facilities for remote targets. These modifications if used in conjunction with the JPDA framework can support the property of run-time evolution that we described in Section 2.

.NET As with Java, the main remote debugging solution for .NET provided through visual studio [Mic12a] pre-purposes a dedicated debugging deployment. In the developer's end the model of the running application is again static, with the developer being responsible for providing the right sources and configuration files. In the case of .NET though the debugger can attach to a running remote process without loosing the context, provided that the static model for the application is available. Although the model in the developer's end is static, a limited form of updating is provided in the form of *edit-and-continue* [Mic12b] of pre-existing methods. There is no support for incremental updating of the target application with new packages, classes or methods.

GDB For Obj-C remote debugging is provided through the gnu-debugger [RS03]. Gdb uses a dedicated process on the target machine called the *gdb-server* to attach to running processes. For full debugging support though the deployed application has to be specifically compiled and deployed with debugging meta-information embedded on the executable which cannot be discarded without re-deployment and loss of the running context. The model for the application on the developer's end is static and depends on the availability of source files. Gdb supports a limited form of updating through an *edit-and-continue* process of pre-existing methods by patching the executable on memory [RS03].

Smalltalk The most prominent example of an interactive debugger is the Smalltalk debugger [BDN+09, LP90]. In Smalltalk the execution context after a failure is never lost since through reflection the debugger can readily be spawned as a separate process and access the environments' reifications for: *processes, exceptions, contexts* etc. Moreover it supports incremental updating in such a way that introducing new behavior through the debugger is

not only possible but is actually advised [BDN+09]. Indeed incremental updating through debugging encourages and supports agile development processes, and more specifically Test Driven Development (TDD) [ABF05]. In addition both the debugging and the reflecting facilities of Smalltalk are extensible. On the one hand the debugger model is written itself in Smalltalk. On the other hand the Smalltalk MOP is readily editable from within the system itself. Illustrative examples of MOP extensions in Smalltalk are given from Rivard in [Riv96].

Bifrost Finally in Smalltalk supporting advanced debugging techniques through instrumentation is illustrated in the Bifrost reflection framework [RRGN10] and through object-centric debugging [RBN12]. Bifrost is an extension to the Smalltalk MOP that relies on explicit meta-objects to provide sub-method [DDLM07] and partial behavioral reflection [TNCC03]. Bifrost is implemented through dynamic re-compilation of methods. Method invocations are intercepted using the *reflective method* abstraction [Mar06] and are subsequently recompiled using AST meta-objects that control the generated bytecode. With Bifrost intercession techniques such as the explicit interception of variable access, is made available at the instance level.

3.2 Comparison

In this Section we compare existing solutions in terms of *run-time evolution,semantic instrumentation and adaptable distribution*.

3.2.1 Run-Time Evolution

In Table 1 we do a comparison in terms of run-time evolution and its sub-properties as there were defined in Section 2:

	JPDA	.NET	GDB	DCE	JREBEL	ST-80	BIFROST
Add/Rem Packages	×	×	×	✓	✓	✓	✓
Add/Rem Classes	×	×	×	✓	✓	✓	✓
Add/Rem IVs	×	×	×	✓	✓	✓	✓
Add/Rem Methods	×	×	×	✓	✓	✓	✓
Method (Body) HotSwapping	✓	✓	✓	✓	✓	✓	✓
Hierarchy Editing	×	×	×	✓	✓	✓	✓

Table 1 – Evaluation on state-of-the-art debugging solutions in terms of run-time evolution

As we see in Table 1 debugging environments of mainstream OO languages (JPDA, .Net Debugger, Gdb) do not support run-time evolution with the exception of a *save-and-continue* facility for pre-existing methods. In the case of Gdb method hotswapping can lead to inconsistencies [Zel05] since it is supported through memory patching, which is a blind process that replaces execution instructions in memory, without knowledge of the underlying semantics of the language. In the Java world recent developments (through Jrebel and DCE) provide full support for this property as does Smalltalk and its extension Bifrost.

3.2.2 Semantic Instrumentation

In Table 2 we do a comparison in terms of semantic instrumentation and its sub-properties as they were defined in Section 2. We have also included a last category marked as *condition/action* that describes whether in all instrumentation events the debugging solution can support user-generated checks and code in order to provide a more fine-grain control. As an example we can consider a conditional breakpoint that is able to execute user specified actions when triggered.

As we can see from our comparison, Bifrost is the front-runner of instrumentation with all other solutions supporting only plain breakpoints and watchpoints. Bifrost lacks an

	JPDA	.NET	GDB	DCE	JREBEL	ST80	BIFROST
Method Execution	✓	✓	✓	✓	✓	✓	✓
Statement Execution	✓	✓	✓	✓	✓	✓	✓
Field Read	✓	×	×	✓	✓	×	✓
Field Write	✓	×	×	✓	✓	×	✓
Object Read	×	✓	✓	×	×	×	✓
Object Write	×	✓	✓	×	×	×	✓
Object Send	×	×	×	×	×	×	✓
Object Receive	×	×	×	×	×	×	✓
Object as Argument	×	×	×	×	×	×	✓
Object Creation	×	×	×	×	×	×	✓
Object Interaction	×	×	×	×	×	×	✓
Object Stored	×	×	×	×	×	×	×
Condition/Action	×	×	✓	×	×	✓	✓

Table 2 – Instrumentation evaluation on state-of-the-art debugging solutions

Object Stored event which is useful for following an object's reference propagation and counting. Finally both Bifrost and Gdb provide support for both conditions and actions on instrumentation events.

3.2.3 Adaptable Distribution

In Table 3 we do a comparison in terms of distribution. Solutions are marked with - for not supporting distribution, + for supporting distribution through a fixed-middleware, ++ for an extensible middleware and +++ for an adaptable middleware.

	JPDA	.NET	GDB	DCE	JREBEL	ST80	BIFROST
Distribution	+	++	+	+	+	-	-

Table 3 – Distribution evaluation on state-of-the-art debugging solutions

As we can see in Table 3 no solution supports an adaptable middleware. The .NET debugging framework leads the comparison using a general purpose and extensible communication solution (DCOM) [Mic13]. We should note here that in the case of Smalltalk (which does not support the property of distribution), there were some efforts in the past to support remote development (including debugging) in Cincom Smalltalk, which were discontinued.

3.2.4 Comparison overview

In Table 4 we present an overview of our comparison in terms of all properties that were described in Section 2:

Property	JPDA	.NET	GDB	DCE	JREBEL	SMALLTALK	BIFROST
Run-Time Evolution	**+ (1/6)**	**+ (1/6)**	**+ (1/6)**	**+++ (6/6)**	**+++ (6/6)**	**+++ (6/6)**	**+++ (6/6)**
Sem. Instrumentation	**+ (4/13)**	**+ (4/13)**	**+ (5/13)**	**+ (4/13)**	**+ (4/13)**	**+ (3/13)**	**+++ (12/13)**
Ad. Distribution	**+ (fixed)**	**++ (extensible)**	**+ (fixed)**	**+ (fixed)**	**+ (fixed)**	**- (no)**	**- (no)**

Table 4 – Summary comparison of state-of-the-art debugging solutions

As we can see from Table 4 debugging solutions based on reflection (such as Smalltalk and Bifrost in the local scenario) offer the most complete solutions in terms of run-time evolution and semantic instrumentation, but lack support for adaptable distribution. On the other hand solutions of mainstream OO languages (JPDA, .Net Debugger, Gdb) and their extensions (Jrebel, DCE) lack support for either run-time evolution or instrumentation (or in some cases both). There is no solution that meets all our criteria in a satisfactory way.

4 Our Solution: Mercury

Our solution proposes a model of the debugged application (cf. Figure 3 (1)) that is dynamic and acts as a meta-level for target applications. It relies on specific meta-objects known as *Mirrors* defined by Bracha and Unghar as "*intermediary objects [...] that directly correspond to language structures and make reflective code independent of a particular implementation*" [BU04]. Mirrors are located on the developer's side. They are causally connected to the debugged application and support *run-time evolution*. The run-time debugging support on the target (cf. Figure 3 (2)) reifies the underlying execution environment to support *semantic instrumentation*. Finally our middleware follows a modular architecture to be adaptable even during runtime (cf. Figure 3 part (3)).

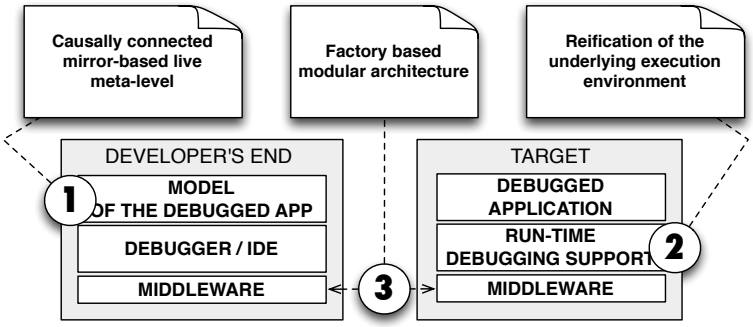


Figure 3 – Overview of our Solution

The rest of this section describes our solution based on this *live meta-level* and how it supports the three properties which were identified in Section 2.

4.1 The Core Meta-Level

The meta-level located on the development machine (left part of Figure 4) is a set of mirrors that reflect on objects (e.g instance of the class Point) on the target side (right part of Figure 4). The target machine also includes support for reflection and debugging. This is the role of the package RTSupport that includes the RunTimeDebuggingSupport class (our remote facade).

On the left side of Figure 4, we depict the 3 core classes of our meta-level. The root is the Mirror class, that declares the targetObject field. So, every mirror holds a remote reference to one object on the target. Nevertheless, an object on the target can be reflected by multiple mirrors on the development side.

Both on the developer's end and on the target, a unique object is responsible to handle all communications to the other side. This object is an instance of RunTimeMirror on the developer's end and an instance of RunTimeDebuggingSupport on the target. On the developer's end, all mirrors can retrieve this object in their inherited field named rtMirror. The API of the RunTimeMirror is completely equivalent to the one of RunTimeDebuggingSupport with one crucial difference: *each call to the target side results in a mirror or a collection of mirrors being returned to the developer's side* ensuring the encapsulation of the remote debugging facilities. This *cascading encapsulation* between calls to a mirror object is equivalent to the transitive wrapping mechanism described by C. Teruel et al. [TCD13] for proxies, only in this case an entire remote environment is being wrapped rather than a local object-graph.

To show how communication and reflection is handled between the development machine and the target, consider the example of the mirror mirrorOnAPoint and its target object aPoint

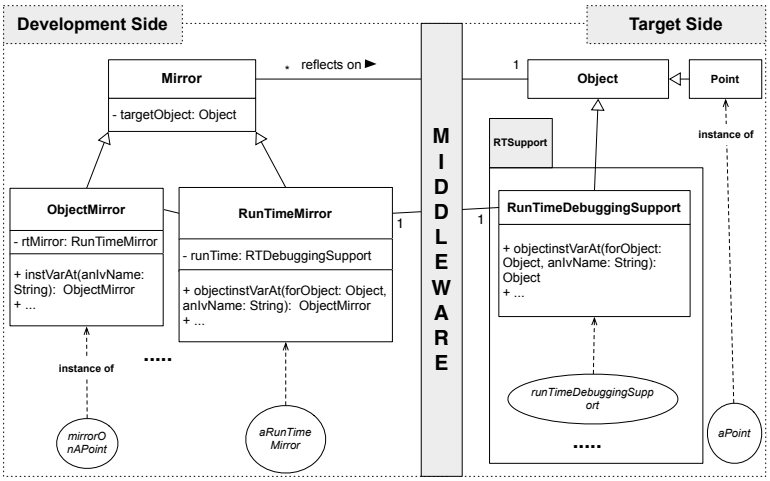


Figure 4 – Our core model

on Figure 5. Suppose that the developer wants to get the class of aPoint. To perform this operation, the IDE sends the getClass message to mirrorOnAPoint. As a result, mirrorOnAPoint sends the getClass(targetRef) message to aRunTimeMirror passing as a parameter the remote reference that it holds. Then, aRunTimeMirror invokes through the middleware the corresponding getClass(targetRef) method on runTimeDebuggingSuppport located on the target. The runTimeDebuggingSuppport retrieves the class Point and answers it back through the middleware. The class on the target is retrieved either via local reflection or through direct vm-support provided by the RunTimeDebuggingSupport class of Figure 4. On the developer side, aRunTimeMirror receives a remote reference on the Point class, and creates a new mirror on the remote class. It is this mirror on the Point class that is returned back to mirrorOnAPoint.

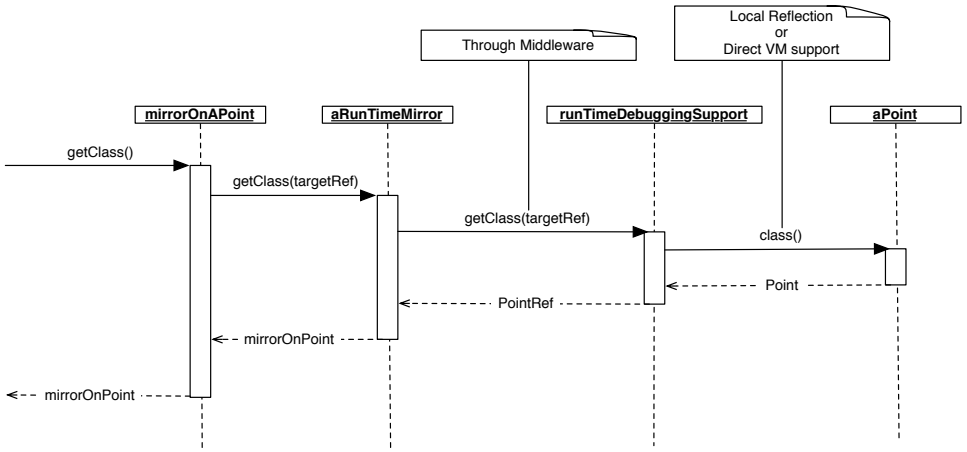


Figure 5 – Sequence Diagram detailing Remote Reflection with Mercury

Communication can also be initiated by the target as shown in Figure 6 to trigger updates of the remote meta-level on the developer's side. For example when a new exception is thrown on the target (right side of Figure 6) the asynchronous newException(exceptionRef) message is send carrying a remote reference to this newly raised exception. As before

aRunTimeMirror will receive the remote reference and will create a new exception mirror. It is this exception mirror that will be forwarded to intrested listeners who have registered through the registerListener(aListener) message of the run-time mirror. Typically a listener will be an instance of the EnviromentMirror class (see Figure 7) through which interrupted processes and unhandled exceptions are accessed.

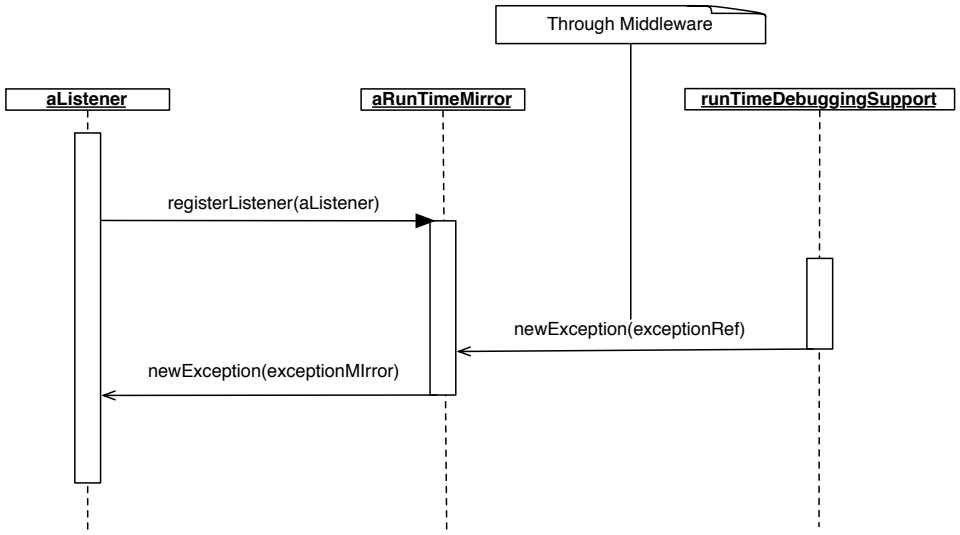


Figure 6 – Asynchronous communication initiated by the target

4.2 Run-Time Evolution

To support run-time evolution, the model of the debugged application on the developer's end and the state of the debugged application on the target (cf. Figure 2) needs to be *causally connected*. This means that an arbitrary change in either one of them should update the other.

We describe how our model supports this property through the class hierarchy and the API of our meta-level (starting from ObjectMirror). Figure 7 depicts 8 core classes of our meta-level which are divided into two groups: the ones that reify the structure of the debugged application (*structural reflection*) and the ones that reify the computation (*computational reflection*) [Fer89, Mae87].

In our model, both *structural reflection* and *computational reflection* are *causally connected* to the other side. For *structural reflection*, this means that the addition of a new package, a new class or method through mirrors, etc. in the development side results in a structural update of the running application on the other side. These 8 core classes depicted in Figure 7 define an API that supports run-time evolution. Instances of these classes reflect on remote objects on the target and all of their methods can be executed while the application is running.

ObjectMirror. An ObjectMirror enables retrieving information from the object reflected such as its class, reading/setting its fields or sending new messages to it but also changing its class (setClass).

EnvironmentMirror. It is the entry point mirror to the target application depicting the remote environment as a whole. Through the environment mirror globals are read/written, loaded packages are retrieved, interrupted processes and unhandled exceptions are

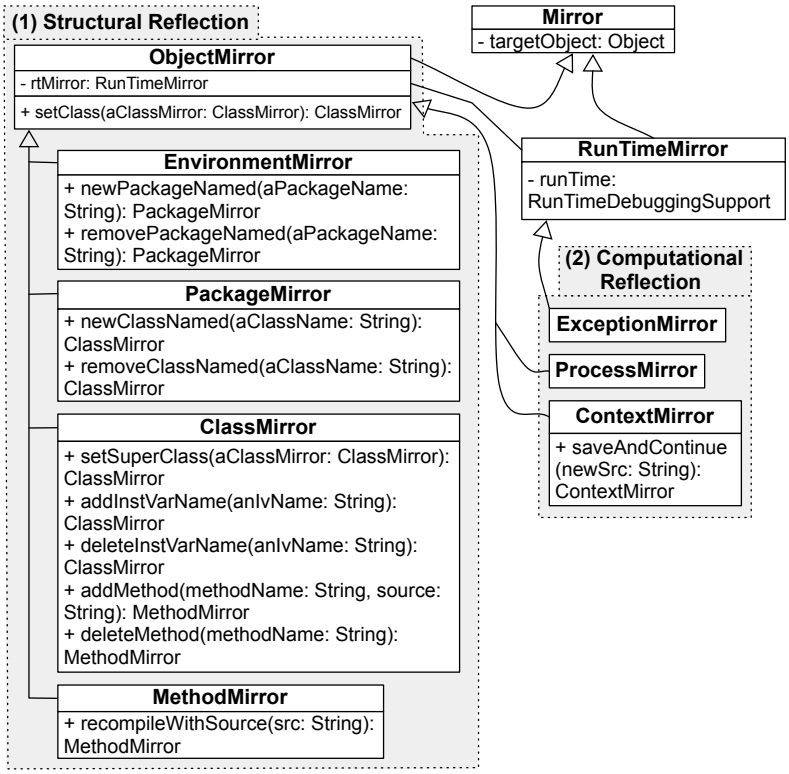


Figure 7 – Core classes and API for supporting Run-Time Evolution

accessed, code is evaluated (evaluate) and packages can be created, removed, or edited (newPackage, removePackage, etc.).

PackageMirror. A package mirror reflects on loaded packages on the target application. This mirror gives access to package's meta-information such as its name and the classes it contains. Classes can also be added or removed using the methods newClassNamed and removeClassNamed.

ClassMirror. Through a class mirror the name, superclass, fields, methods and enclosing package of the reflected class can be retrieved. The superclass can be changed, new instance variables and methods can be added/removed or edited (setSuperClass, addInstVarName, deleteInstVarName, addMethod, deleteMethod, etc.).

MethodMirror. Apart from retrieving the name, source or class membership of a Method, the developer can edit a method in place (recompileWithSource).

ProcessMirror. It allows one to retrieve meta-information on a process such as its stack and manipulate the execution flow.

ExceptionMirror. It is the reification of exceptions on the target. Through an exception mirror the description of an unhandled exception can be retrieved, as well as the process that it occurred and the offending execution context.

ContextMirror. It is the reification of a stack frame (context) on the target application. Through a contextMirror its process, method, receiver and sender can be retrieved,

temporaries and arguments of the invocation can be read/written, its execution can be restarted but also the method that was invoked and created the context can be edited before continuing the execution (saveAndContinue).

4.3 Semantic Instrumentation

Semantic Instrumentation in our model is supported through *intercession*. Specifically the underlying execution environment is reified inside the run-time environment of the target as to be able to control the semantics of a running process.

The model of our solution uses the following patterns:

The observer [ABW98] An observer defines a dependency between an object and its *dependents*, so that the dependents are notified for state changes on that object.

The implicit meta-object [Mae87] Implicit meta-objects are meta-objects that are invoked automatically by the underlying execution mechanism.

Objects can be instrumented either to perform user-generated conditions and actions upon invocation of specific events (e.g RunTimeDebuggingSupport»objectOnReceive) or to halt the process on those specific events (e.g RunTimeDebuggingSupport»objectHaltOnReceive).

Figure 8 depicts the reification of the Interpreter (the underlying execution environment) which acts as our observer, connecting instances of Object (regular objects) to instances of ImplicitMetaObject (dependents). Whenever an event of interest is being applied to an object (such as a message send) the underlying execution mechanism invokes the Interpreter reification, which in turn notifies the ImplicitMetaObjects. The Interpreter resolves the relationship between objects and meta-objects through the MetaEnvironment, which acts as an environment dictionary for the meta-level. The MetaEnvironment provides a one-to-one mapping between objects and meta-objects.

Implicit meta-objects when notified, will invoke a callback (class Closure in Figure 8) which can be either a local callback or a remote callback from the developer's end. The RunTimeDebuggingSupport maintains a reference to the Interpreter reification to register these callbacks coming from mirrors on the developer's side.

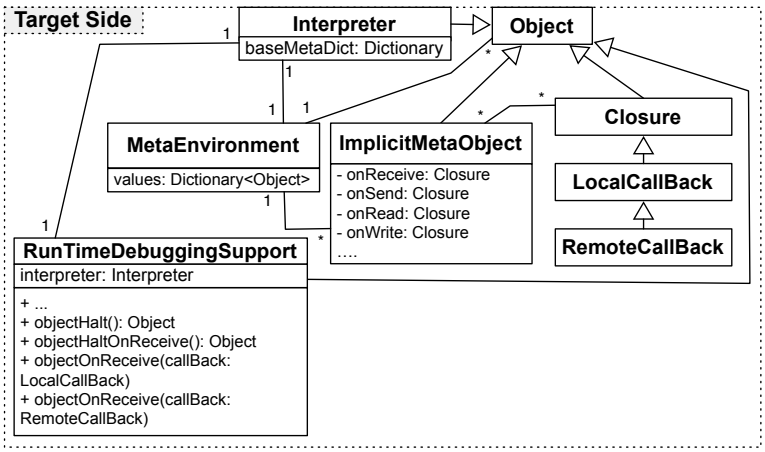


Figure 8 – Core classes for Instrumentation support in the Target

4.4 Adaptable Distribution

To support distribution via an adaptable middleware, we modeled our solution using the concept of the abstract Factory [ABW98], through which families of related objects can be assembled and parametrized at runtime.

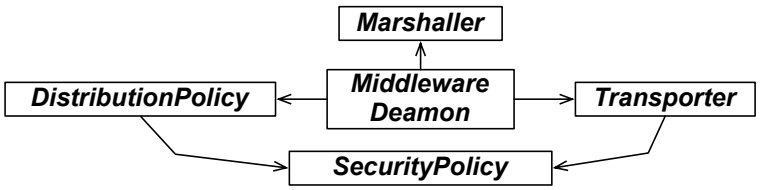


Figure 9 – Core classes of our adaptable middleware

Figure 9 depicts the core classes of our model for distribution:

Middleware Deamon This abstract class defines methods for the orchestration (assembling) and initialization of our middleware, acting as an Abstract Factory. It is also responsible for loading the RTSupport package (cf Figure 4) on the target upon the successful authentication of a client.

Transporter The concrete subclasses of this abstract class handle the actual communication between peers. Different transporters can support different communication protocols (e.g tcp, udp or web-sockets)

Marshaller The marshaller (through its concrete subclasses) is responsible for serializing and materializing information, passed through the connection. Different marshallers can support different transcoding algorithms to fit the needs of the debugging context (e.g serializing to xml, json or binary-form).

Distribution Policy This class (through its concrete subclasses) decides how specific objects or group of objects will be distributed among peers. Options can include: full serialization, shallow serialization, proxying, etc..

Security Policy The concrete subclasses of this abstract class are responsible for authentication and for restricting access (either message sending or distribution) for specific instances or whole classes of objects.

4.5 Comparison with Existing Solutions

In this Section we compare the state-of-the-art debugging solutions (which we discussed in Section 3.2.4) with our work in terms of *run-time evolution, semantic instrumentation and adaptable distribution*.

As we can see from Tables 5 and 6 our solution manages to cover all three properties that were identified in Section 2 being comparable only to the Bifrost framework (in the local scenario) in terms of run-time evolution and semantic instrumentation. In our case though these properties are brought to remote debugging through an adaptable middleware. In terms of distribution Mercury is only comparable to the .NET debugging framework which uses a general purpose extensible (but not adaptable) communication middleware (DCOM) [Mic13].

Finally since both our solution and Bifrost are based on Smalltalk, we were also able to perform a micro-benchmark to compare the two, in terms of the overhead introduced by instrumentation. The benchmark is based on Tanter [TNCC03] and the Bifrost metrics

Property		JPDA	.NET	GDB	DCE	JREBEL	ST-80	BIFROST	MERCURY
Run-Time Evolution	Add/Rem Packages	✗	✗	✗	✓	✓	✓	✓	✓
	Add/Rem Classes	✗	✗	✗	✓	✓	✓	✓	✓
	Add/Rem IVs	✗	✗	✗	✓	✓	✓	✓	✓
	Add/Rem Methods	✗	✗	✗	✓	✓	✓	✓	✓
	Method (Body) HotSwapping	✓	✓	✗	✓	✓	✓	✓	✓
	Hierarchy Editing	✗	✗	✗	✓	✓	✓	✓	✓
Sem. Instrumentation	Method Execution	✓	✓	✓	✓	✓	✓	✓	✓
	Statement Execution	✓	✓	✓	✓	✓	✓	✓	✓
	Field Read	✓	✗	✗	✓	✓	✗	✓	✓
	Field Write	✓	✗	✗	✓	✓	✗	✓	✓
	Object Read	✗	✗	✗	✗	✗	✗	✓	✓
	Object Write	✗	✗	✗	✗	✗	✗	✓	✓
	Object Send	✗	✗	✗	✗	✗	✗	✓	✓
	Object Receive	✗	✗	✗	✗	✗	✗	✓	✓
	Object as Argument	✗	✗	✗	✗	✗	✗	✓	✓
	Object Creation	✗	✗	✗	✗	✗	✗	✓	✓
	Object Interaction	✗	✗	✗	✗	✗	✗	✓	✓
	Object Stored	✗	✗	✗	✗	✗	✗	✗	✓
	Condition/Action	✗	✗	✓	✗	✗	✓	✓	✓
Ad. Distribution		+	++	+	+	+	-	-	+++

Table 5 – Properties evaluation of Mercury and existing debugging solutions

Property	JPDA	.NET	GDB	DCE	JREBEL	SMALLTALK	BIFROST	MERCURY
Run-Time Evolution	**+ (1/6)**	**+ (1/6)**	**+ (1/6)**	**+++ (6/6)**	**+++ (6/6)**	**+++ (6/6)**	**+++ (6/6)**	**+++ (6/6)**
Sem. Instrumentation	**+ (4/13)**	**+ (4/13)**	**+ (5/13)**	**+ (4/13)**	**+ (4/13)**	**+ (3/13)**	**+++ (12/13)**	**+++ (13/13)**
Ad. Distribution	**+ (fixed)**	**++ (extensible)**	**+ (fixed)**	**+ (fixed)**	**+ (fixed)**	**- (no)**	**- (no)**	**+++ (adaptable)**

Table 6 – Comparison of Mercury with existing solutions

are those reported in [Res12]. The benchmark measures the slowdown introduced by each solution for one million messages send to a test object when a) no instrumentation is present b) instrumentation is loaded but is disabled for this specific object and c) instrumentation is enabled on the test object of the micro-benchmark.

	BIFROST	MERCURY
No instrumentation	1x	1x
Disabled instrumentation	1x	1x
Enabled instrumentation	35x	8x

Table 7 – Instrumentation benchmark for Bifrost and Mercury

As we see in Table 7 for both solutions there is no overhead introduced when a specific object is not being instrumented, regardless of whether the solution is loaded into the environment. This is important for practical reasons so as to avoid slowing down the whole system while debugging. While instrumenting a specific object our solution introduces a significantly smaller overhead than Bifrost. We believe that this is due to the fact that our solution is based on the underlying virtual-machine rather than on byte-code manipulation as in the case of

Bifrost.

5 Mercury's Implementation

5.1 Implementation Overview

Given our state-of-the-art survey in Section 3 we chose to implement Mercury in a platform that was as close as possible to our goals. We chose to implement a prototype[1] of our model (described in Section 4) in Pharo [BDN+09] and Slang [IKM+97]. Pharo is a reflective, object-oriented and dynamically typed programming environment that is inspired by Smalltalk. Slang is a subset of the Smalltalk syntax with procedural semantics that can be easily translated to C. In Figure 10 we show the different constituents of our implementation.

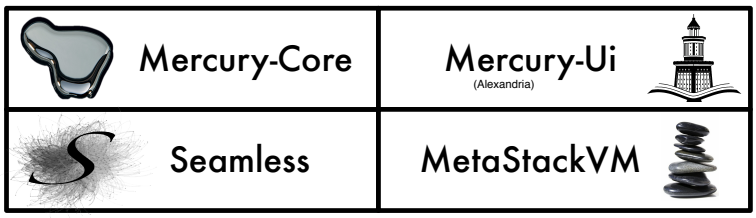


Figure 10 – Core parts of Mercury's Prototype

MetaStackVM Is a dedicated virtual-machine for debugging targets, that extends the reflective facilities of the standard Stack VM of Pharo [Mir08] in order to support intercession [Pap13].

Seamless Is our adaptable middleware that provides flexible communication facilities betweens peers during debugging sessions.

Mercury-Core Is the sub-project of Mercury that hosts the debugging meta-level and the debugging run-time support.

Mercury-UI Is a debugging front-end that exemplifies key functionalities of our solution.

All four part of our prototype implementation for Mercury are released under the MIT license [2].

5.2 Discussion: Implementation trade-offs

5.2.1 Supporting Run-Time Evolution

Implementors of our model have essentially two options for supporting run-time evolution through the RunTimeDebuggingSupport (depicted in the left side of Figure 4):

(a) Local reflection Local reflection on the target can be used to provide the corresponding API for run-time evolution. This solution is applicable to languages that already provide a rich set of local reflective facilities. It is also a portable and extensible solution since the debugging support is written in the same language as the target application.

[1] http://ss3.gemstone.com/ss/Mercury-Prototype.html
[2] http://opensource.org/licenses/MIT

(b) Virtual Machine support Debugging support on the target can be also hard-coded inside the virtual-machine of the target. This solution fits better with languages that do not support advanced reflective facilities on their own. It is also an attractive option for system debugging, in cases where core language reflection itself has to be debugged. This solution is less portable and extensible if it is not supported by the vendor of the target language.

In our prototype we used a combination of the two approaches mentioned above. Remote reflection on the instance level is separated from local reflection on the target and can thus support some limited form of system debugging. However, we also make use of local reflective facilities on the target for system-organization reflection (packaging meta-objects) and computational reflection (reifications of contexts and processes). Our implementation currently depends on the compiler on the target. Ideally, the developers' end compiler should be used and the target should not host a compiler itself to further minimize the footprint.

5.2.2 Supporting Instrumentation

To support semantic instrumentation the following options can apply:

(a) Bytecode Manipulation The compiler can be used to re-compile part of the system to transparently introduce crosscuts that perform instrumentation checks (for message sending, field access, etc.). This solution has the disadvantage of instrumenting only static entities (such as classes or methods) and may perform poorly when specific objects (runtime entities) need to be instrumented. For example when instrumenting message sending on a specific object, all the methods of its class and its superclasses have to be re-compiled to introduce the crosscuts. On the other hand in the case of a self-hosted compiler this option favors portability.

(b) Virtual Machine support Instrumentation support on the target can be also hard-coded inside the virtual-machine of the target. This solution fits better with instrumentation of run-time entities, since the checking can be performed on the object itself while it is being interpreted by the underlying execution environment. Portability may be an issue in this case if instrumentation is not supported by the vendor.

In our prototype we supported instrumentation by extending the stack-based virtual machine of Pharo. We chose to provide virtual-machine support since our focus was on instrumenting run-time rather than static entities. Furthermore we did not wish to have further dependencies on the compiler of the target.

5.3 Discussion: Implementing Mercury in Java

We take the Java language as an example to investigate the feasibility of implementing Mercury in other languages. We discuss the technological prerequisites for implementors for each part of our model as was discussed in Section 4.

A causally connected dynamic meta-level for debugging that can support run-time evolution can be build for Java by combining the currently available debugging infrastructure found in JPDA [Ora13b] with the incremental updating facilities of the DCE VM project [WWS10] or those found in the JRebel vm-plugin [Zer12] (see also Section 3). Support for semantic instrumentation can be build on top of solutions for bytecode manipulation or reflective intercession for Java like those in Iguana/J [RC00], Reflex [TBN01], ASM [BLC02] or JavaAssist [CN03]. We should note here however that since Java is more static in nature - compared

to Pharo - these frameworks should be able to inter-operate with the incremental updating support we discussed previously (DCE, JRebel) to apply the required adaptations at run-time as we do with Mercury. Supporting remote debugging through an adaptable middleware can be achieved in Java by substituting the static low-level debugging communication protocol (JDWP) of JPDA with a more dynamic and flexible middleware solution like Cajo [Cat14].

6 Mercury's Validation

For validating Mercury we have considered three different kinds of constraint devices as debugging targets. These devices (see Figure 11) where chosen as illustrative examples of either:

- Targets that have different hardware or environment settings than development machines.
- Targets that are not locally or easily accessible.
- Targets that have resource constraints or no input/output interfaces for local development.

Through this setting, we have verified the applicability of Mercury for different debugging targets. We experimented on how a debugging session can benefit from Mercury's properties, by studying the following two use-cases:

1. Combining agile development [ABF05] with debugging **in a single remote debugging session** without the need of re-deployment.
2. Supporting both OO-centric [RBN12] and Stack-based debugging **in a remote setting** through remote object instrumentation.

Figure 11 shows the set-up of our experiment. In the upper part of the figure we depict our debugging targets. Device A is a smart-phone target connected to our development machine through wifi. Device B is a tablet target also connected through a wireless network, while Device C is a remote server to which we connect through ethernet.

In the lower part of the figure we show the development machine running our debugging front-end. A cropped screenshot of the Mercury IDE is shown at the center of the figure. Each tab corresponds to tools supporting remote development and debugging of a single target. The developer machine connects to our targets through the two communication interfaces designated as *ETH and WIFI* for ethernet and wireless communication channels respectively. For our two android devices (phone and tablet), we have also tested communication through a usb channel that establishes ethernet connections using port forwarding. [3]

With this setting we have been able to achieve the following:

1. Re-produce an initial error multiple times in order to test different hypothesis without the need of re-deployment.
2. Simplify offending contexts without re-starting the debugging session.
3. Maintain the state and suspended execution flow of initial unhandled errors:
   (a) In order to cross-examine the initial failing state with new findings.
   (b) In case the initial errors are not easily reproducible (as is the case with heisenbugs [Gra86])

[3] http://developer.android.com/tools/help/adb.html

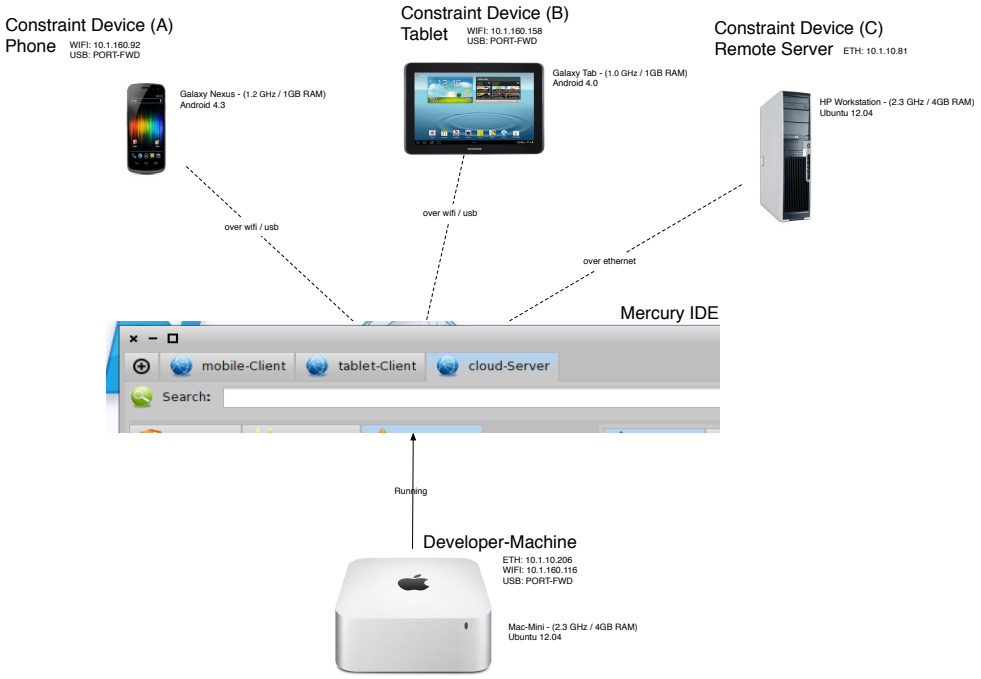


Figure 11 – Experimental Set-up for our Debugging Targets

4. Verify that the remote object instrumentation facilities of Mercury can bring the idea of oo-centric debugging in a distributed setting.

5. Provide an example where Mercury uses the two paradigms of stack-based and oo-centric debugging in a complementary fashion.

7 Conclusion and Future Work

In this work we have proposed Mercury: a live mirror-based model and infrastructure for remote debugging. Mercury exhibits three desirable properties that we have identified as important for remote debugging, namely: *run-time evolution*, *semantic instrumentation*, and *adaptable distribution* Run-time evolution is the ability of a remote debugging solution to incrementally update all parts of a remote application without losing the running context (i.e without stopping the application). Semantic instrumentation is the ability of a debugging solution to alter the semantics of a running process to assist debugging. Finally, adaptable distribution is the ability of a debugging solution to adapt its underlying middleware while debugging a remote target.

Mercury supports run-time evolution through a causal connection between the meta-level running on the developer machine, and the application to debug (the base-level) on the target device. The two levels are connected both computationally and structurally. It supports semantic instrumentation through the reification of the underlying execution environment (virtual-machine) inside the run-time environment of the target (as an interpreter). Finally adaptable distribution is supported through a modular architecture of the underlying middleware. We have validated the applicability of our proposal through a prototype implementation

in the Pharo language. We have illustrated our approach through several working examples in an experimental setting of two case-studies.

Future Work A future perspective for this work is to examine the prerequisites of supporting advanced debugging facilities such as delta-debugging [Zel02] in a remote setting. We also plan to explore more issues of mirror-based systems in a remote setting such as ontological correspondence [BU04]. Finally we would like to extend our implementation to be completely independent from local reflection facilities on the target (such as the host compiler) as exemplified in our previous work with MetaTalk [PBD+11].

References

[ABF05] Alex Abacus, Mike Barker, and Paul Freedman. Using test-driven software development tools. *IEEE Software*, 22(2):88–91, 2005.

[ABW98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, Boston, MA, USA, 1998.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009. URL: http://pharobyexample.org/.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Proceedings of Adaptable and Extensible Component Systems*, Grenoble, France, November 2002.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press. URL: http://bracha.org/mirrors.pdf.

[Cat14] John Catherino. The cajo project. https://java.net/projects/cajo/pages/Home, 2014.

[CN03] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In *In Proceedings of the second International Conference on Generative Programming and Component Engineering (GPCE'03)*, volume 2830 of *LNCS*, pages 364–376, 2003.

[DDLM07] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007. URL: http://rmod.lille.inria.fr/archives/papers/Denk07b-TOOLS07-Submethod.pdf.

[DL02] Pierre-Charles David and Thomas Ledoux. An infrastructure for adaptable middleware. In *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 773–790. Springer Berlin Heidelberg, 2002. URL: http://dx.doi.org/10.1007/3-540-36124-3_52.

[Fer89] Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 317–326, October 1989.

[Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, November 1997. URL: http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/squeak.html, doi:10.1145/263700.263754.

[LP90] Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk: vol. 1*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.

[Mar06] Philippe Marschall. Persephone: Taking Smalltalk reflection to the sub-method level. Master's thesis, University of Bern, December 2006. URL: http://scg.unibe.ch/archive/masters/Mars06a.pdf.

[McA95] Jeff McAffer. Meta-level programming with coda. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 190–214, Aarhus, Denmark, August 1995. Springer-Verlag.

[Mic12a] Microsoft. How to: Set up remote debugging, visual studio 2012. http://msdn.microsoft.com/en-us/library/bt727f1t.aspx, 2012.

[Mic12b] Microsoft. Supported code changes (c#), visual studio 2012. http://msdn.microsoft.com/en-us/library/ms164927.aspx, 2012.

[Mic13] Microsoft. Setting up remote debugging, visual studio 2013. http://msdn.microsoft.com/en-us/library/bt727f1t%28v=vs.71%29.aspx, 2013.

[Mir08] Eliot Miranda. Cog blog. speeding up croquet and squeak with a new open-source vm from qwaq, 2008. URL: http://www.mirandabanda.org/cogblog/.

[Ora13a] Oracle. Java debug interface (jdi). http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html, 2013.

[Ora13b] Oracle. Java platform debugger architecture (jpda). http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/, 2013.

[Pap13] Nikolaos Papoulias. *Remote Debugging and Reflection in Resource Constrained Devices*. These, Université des Sciences et Technologie de Lille - Lille I, December 2013.

[PBD+11] Nikolaos Papoulias, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Towards structural decomposition of reflection with mirrors. In *Proceedings of International Workshop on Smalltalk Technologies (IWST'11)*, Edingburgh, United Kingdom, 2011. URL: http://hal.inria.fr/inria-00629175/en/.

[RBN12] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceeding of the 34rd international conference on Software engineering*, ICSE '12, 2012. URL: http://scg.unibe.ch/archive/papers/Ress12a-ObjectCentricDebugging.pdf, doi:10.1109/ICSE.2012.6227167.

[RC00] Barry Redmond and Vinny Cahill. Iguana/J: Towards a dynamic and efficient reflective architecture for java. In *Proceedings of European Conference on*

Object-Oriented Programming, workshop on Reflection and Meta-Level Architectures, 2000.

[Res12] Jorge Ressia. *Object-Centric Reflection*. PhD thesis, Institut fur Informatik und angewandte Mathematik, 2012.

[Riv96] Fred Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, April 1996.

[RRGN10] Jorge Ressia, Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Run-time evolution through explicit meta-objects. In *Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*, pages 37–48, October 2010. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-641/. URL: http://scg.unibe.ch/archive/papers/Ress10a-RuntimeEvolution.pdf.

[RS03] Stan Shebs Richard Stallman, Roland Pesch. *Debugging with GDB*. Gnu Press, 2003.

[TBN01] Éric Tanter, Noury Bouraqadi, and Jacques Noyé. Reflex — towards an open reflective extension of Java. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 25–43. Springer-Verlag, 2001.

[TCD13] Camille Teruel, Damien Cassou, and Stéphane Ducasse. Object Graph Isolation with Proxies. In *DYLA - 7th Workshop on Dynamic Languages and Applications, Collocated with 26th European Conference on Object-Oriented Programming - 2013*, Montpellier, France, 2013.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003. URL: http://www.dcc.uchile.cl/~etanter/research/publi/2003/tanter-oopsla03.pdf.

[WWS10] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for java. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10. ACM, 2010.

[Zel02] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA, 2002. ACM Press. doi:10.1145/587051.587053.

[Zel05] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005.

[Zer11] ZeroTurnAround. Java ee productivity report 2011. http://zeroturnaround.com/wp-content/uploads/2010/11/Java_EE_Productivity_Report_2011_finalv2.pdf, 2011.

[Zer12] ZeroTurnAround. What developers want: The end of application re-deploys. http://files.zeroturnaround.com/pdf/JRebelWhitePaper2012-1.pdf, 2012.